Chapter 1. Key Technologies for Big Data Stream Computing

Dawei Sun, Guangyan Zhang, Weimin Zheng, and Keqin Li

Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

1.1 Introduction

Big data computing is a new trend for future computing with the quantity of data growing and the speed of data increasing. In general, there are two main mechanisms for big data computing, i.e., big data stream computing and big data batch computing. Big data stream computing is a model of straight through computing, such as Storm [1] and S4 [2] which do for stream computing what Hadoop does for batch computing, while big data batch computing is a model of storing then computing, such as MapReduce framework [3] open sourced by the Hadoop implementation [4].

Essentially, big data batch computing is not sufficient for many real-time application scenarios, where a data stream changes frequently over time, and the latest data are the most important and most valuable. For example, when analyzing data from real-time transactions (e.g., financial trades, email messages, users search requests, sensor data tracking), a data stream grows monotonically over time as more transactions take place. Ideally, a real-time application environment can be supported by big data stream computing. Generally, big data streaming computing has the following defining characteristics [5, 6]. (1) The input data stream is a real-time data stream and needs real-time computing, and the results must be updated every time the data changes. (2) Incoming data arrive continuously at volumes that far exceed the capabilities of individual machines. (3) Input streams incur multi-staged computing at low latency to produce output streams, where any incoming data entry is ideally reflected in the newly generated results in output streams within seconds.

1.1.1 Stream Computing

Stream computing, the long-held dream of "high real-time computing" and "high-throughput computing", with programs that compute continuous data streams, has opened up the new era of future computing due to big data, which is a datasets that is large, fast, dispersed, unstructured, and beyond the ability of available hardware and software facilities to undertake their acquisition, access, analytics, and application in a reasonable amount of time and space [7] [8]. Stream computing is a computing paradigm that reads data from collections of software or hardware sensors in a stream form and computes continuous data streams, where feedback results should be in a real-time data stream as well. A data stream is a sequence of data sets, and a continuous stream is an infinite sequence of data sets, and parallel streams have more than one stream to be processed at the same time.

Stream computing is one effective way to support big data by providing extremely low-latency velocities with massively parallel processing architectures, and is becoming the fastest and most efficient way to obtain useful knowledge from big data, allowing organizations to react quickly when problems appear or to predict new trends in the near future [9] [10].

A big data input stream has the characteristics of high speed, real time, and large volume for applications such as sensor networks, network monitoring, micro blog, web exploring, social networking, and so on. These data sources often take the form of continuous data streams, and timely analysis of such a data stream is very important as the life cycle of most data is very short [8] [11] [12]. Furthermore, the volume of data is so high that there is no enough space for storage, and not all data need to be stored. Thus, the storing-then-computing batch computing model does not fit at all. Nearly all data in big data environments have the feature of streams, and stream computing has appeared to solve the dilemma of big data computing by computing data online

within real-time constraints [13]. Consequently, the stream computing model will be a new trend for high-throughput computing in the big data era.

1.1.2 Application Background

Big data stream computing is able to analyze and process data in real time to gain an immediate insight, and it is typically applied to the analysis of vast amount of data in real time and to process them at a high speed. Many application scenarios require big data stream computing. For example, in financial industries, big data stream computing technologies can be used in risk management, marketing management, business intelligence, and so on. In the Internet, big data stream computing technologies can be used in search engines, social networking, and so on. In Internet of things, big data stream computing technologies can be used in intelligent transportation, environmental monitoring, and so on.

Usually, a big data stream computing environment is deployed in a highly distributed clustered environment, as the amount of data is infinite, the rate of data stream is high, and the results should be real-time feedback.

1.1.3 Chapter Organization

The remainder of this paper is organized as follows. In Section 1.2, we introduce data stream graphs and the system architecture for big data stream computing (BDSC), and key technologies for BDSC systems. In Section 1.3, we present the system architecture and key technologies of four popular example BDSC systems, which are Twitter Storm, Yahoo! S4, and Microsoft TimeStream and Naiad. Finally, we discuss grand challenges and future directions in Section 1.4.

1.2 Overview of a Big Data Stream Computing System

In this section, we first present some related concepts and definitions of directed acyclic graphs

and stream computing. Then, we introduce the system architecture for stream computing and the key technologies for BDSC systems in big data stream computing environments.

1.2.1 DAG and Stream Computing

In stream computing, the multiple continuous parallel data streams can be represented by a task topology, also named a data stream graph, which is usually described by a directed acyclic graph (DAG) [5, 14-16]. A measurable data stream graph view can be defined by Definition 1.

Definition 1. A *data stream graph* G is a directed acyclic graph, which is composed of a vertices set and a directed edges set, and has a logical structure and a special function, and is denoted as $G=\left(V\left(G\right),E\left(G\right)\right)$, where $V\left(G\right)=\left\{v_1,v_2,\cdots,v_n\right\}$ is a finite set of *n* vertices, which represent tasks, and $E\left(G\right)=\left\{e_{1,2},e_{1,3},\cdots,e_{n-1,n}\right\}$ is a finite set of directed edges, which represent data stream between vertices. If $\exists e_{i,j}\in E\left(G\right)$, then $v_i,v_j\in V\left(G\right)$, $v_i\neq v_j$, and $\left\langle v_i,v_j\right\rangle$ is an ordered pair, where a data stream comes from v_i and goes to v_j.

The in-degree of vertex v_i is the number of incoming edges, and the out-degree of vertex v_i is the number of outgoing edges. A source vertex is a vertex whose in-degree is zero, and an end vertex is a vertex whose out-degree is zero. A data stream graph G has at least one source vertex and one end vertex.

For the example data stream graph with eleven vertices shown in Figure 1, the vertices set is $V=\left\{v_a,v_b,\cdots,v_k\right\}$, the directed edges set is $E=\left\{e_{a,c},e_{b,c},\cdots,e_{j,k}\right\}$, the source vertices are v_a and v_b, and the end vertex is v_k. The in-degree of vertex v_d is one, and the out-degree of vertex v_d is two.

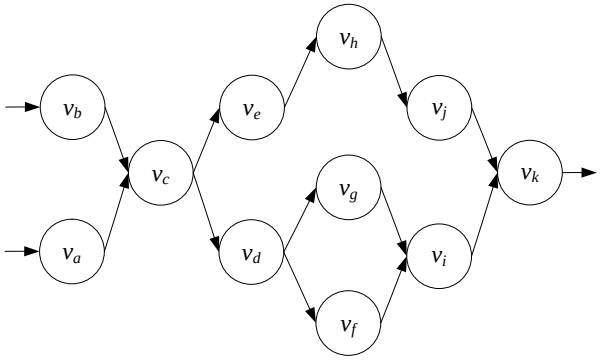

Figure 1. A data stream graph

Definition 2. A *sub-Graph sub-G* of the data stream graph G is a sub-graph consisting of a subset of the vertices with the edges in between. For vertices v_i and v_j in the sub-Graph *sub-G* and any vertex *v* in the data stream graph G, *v* must also be in the *sub-G* if *v* is on a directed path from v_i to v_j, that is $\forall v_i, v_j \in V(sub\text{-}G)$, $\forall v \in V(G)$, if $v \in V\left(p(v_i, v_j)\right)$, then $v \in V\left(p(sub\text{-}G)\right)$.

A sub-Graph *sub-G* is logically equivalent and can be substituted by a vertex. But reducing that sub-Graph to a single logical vertex would create a graph with cycle, not a DAG.

Definition 3. A *path* $p(v_u, v_v)$ from vertex v_u to vertex v_v is a subset of $E\left(p(v_u, v_v)\right)$, which should meet the conditions: $\exists e_{i,k} \in p(v_u, v_v), e_{k,j} \in p(v_u, v_v)$, for any directed edge $e_{k,l}$ in path $p(v_u, v_v)$ displays the properties: if $k \neq i$, then $\exists m$, and $e_{m,k} \in p(v_u, v_v)$; if $i \neq j$, then $\exists m$, and $e_{l,m} \in p(v_u, v_v)$.

The latency $l_p(v_u, v_v)$ of a path from vertex v_u to vertex v_v is the sum of latencies of both vertices and edges on the path, as given by (1):

$$l_p(v_u, v_v) = \sum_{v_i \in V\left(p(v_u, v_v)\right)} c_{v_i} + \sum_{e_{i,j} \in E\left(p(v_u, v_v)\right)} c_{e_{i,j}}, c_{v_i}, c_{e_{i,j}} \geq 0. \quad (1)$$

A critical path, also called the longest path, is a path with the longest latency from a source

vertex v_s to an end vertex v_e in a data stream graph G, which is also the latency of data stream graph G.

If there are m paths from source vertex v_s to end vertex v_e in data stream graph G, then the latency $l(G)$ of data stream graph G is given by (2):

$$l(G)=\max\left\{l_{p_1}(v_s,v_e),l_{p_2}(v_s,v_e),\cdots,l_{p_m}(v_s,v_e)\right\}. \tag{2}$$

where $l_{p_i}(v_s,v_e)$ is the latency of the ith path from vertex v_s to vertex v_e.

Definition 4. In data stream graph G, if $\exists e_{i,j}$ from vertex v_i to vertex v_j, then vertex v_i is a *parent* of vertex v_j, and vertex v_j is a *child* of vertex v_i.

Definition 5. The *throughput* $t(v_i)$ of vertex v_i is the average rate of successful data stream computing in a big data environment, and is usually measured in bits per second (bps).

We identify the source vertex v_s as in the first level, the children of source vertex v_s as in the second level, and so on, and the end vertex v_e as in the last level.

The throughput $t(level_i)$ of the ith level can be calculated by (3):

$$t(level_i)=\sum_{k=1}^{n_i}t(v_k), \tag{3}$$

where n_i is the number of vertices in the ith level.

If data stream graph G has m levels, then the throughput $t(G)$ of the data stream graph G is the minimum throughput of all the throughput in the m levels, as described by (4):

$$t(G)=\min\left\{t(level_1),t(level_2),\cdots,t(level_m)\right\}, \tag{4}$$

where $t(level_i)$ is the throughput of the ith level in data stream G.

Definition 6. A *topological sort* $TS(G)=(v_{x_1},v_{x_2},\cdots,v_{x_n})$ of the vertices $V(G)$ in data stream

graph G is a linear ordering of its vertices, such that for every directed edge e_{x_i,x_j} ($e_{x_i,x_j} \in E(G)$) from vertex v_{x_i} to vertex v_{x_j}, v_{x_i} comes before v_{x_j} in the topological ordering.

A topological sort is possible if and only if the graph has no directed cycle, that is, it needs to be a directed acyclic graph. Any directed acyclic graph has at least one topological sort.

Definition 7. A *graph partitioning* $GP(G) = \{GP_1, GP_2, \cdots, GP_m\}$ of the data stream graph G is a topological sort based split of the vertex set $V(G)$ and the corresponding directed edges. A graph partitioning should meet the non-overlapping and covering properties, that is, if $\forall i \neq j$, $i, j \in [1, m]$, then $GP_i \cap GP_j = \varnothing$, and $\bigcup_{i=1}^{m} GP_i = V(G)$.

1.2.2 System Architecture for Stream Computing

In big data stream computing environments, stream computing is the model of straight through computing. As shown in Figure 2, the input data stream is in a real-time data stream form, and all continuous data streams are computed in real time, and the results must be updated also in real time. The volume of data is so high that there is no enough space for storage, and not all data need to be stored. Most data will be discarded, and only a small portion of the data will be permanently stored in hard disks.

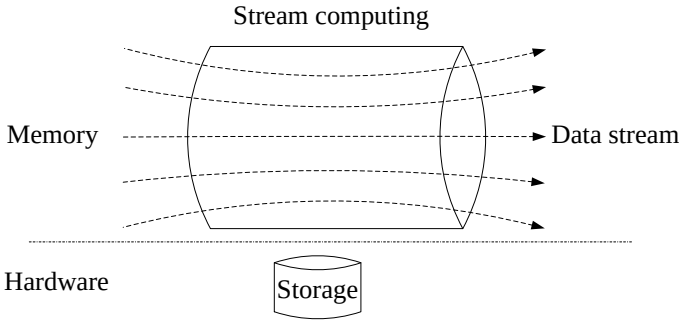

Figure 2. A big data stream computing environment

1.2.3 Key Technologies for BDSC Systems

Due to data streams' distinct features of real time, volatility, burstiness, irregularity, and infinity in a big data environment, a well-designed big data stream computing (BDSC) system always optimizes in system structure, data transmission, application interfaces, high-availability, and so on [17-19].

1.2.3.1 System Structure

Symmetric structure and master-slave structure are two main system structures for BDSC systems, as shown in Figure 3 and Figure 4, respectively.

In the symmetric structure system, as shown in Figure 3, the functions of all nodes are the same. So it is easy to add a new node or to remove an unused node, and to improve the scalability of a system. However, some global functions such as resources allocation, fault tolerance, and load balancing are hard to achieve without a global node. In S4 system, the global functions are achieved by borrowing distributed protocol zookeeper.

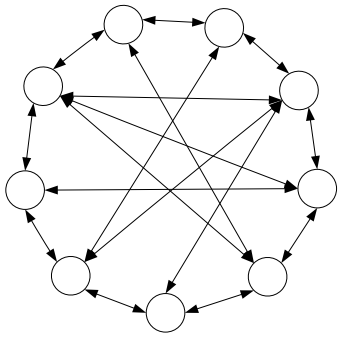

Figure 3. Symmetric structure

In the master-slave structure system, as shown in Figure 4, one node is the master node, and other nodes are slave nodes. The master node is responsible for global control of the system, such as resources allocation, fault tolerance, and load balancing. Each slave node has a special function, and it always receives a data stream from the master node, processes the data stream, and sends the

results to the master node. Usually, the master node is the bottleneck in the master-slave structure system. If it fails, the whole system will not work.

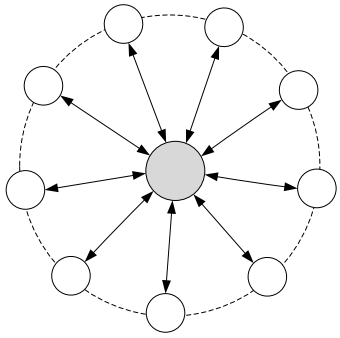

Figure 4. Master-slave structure

1.2.3.2 Data Stream Transmission

Push and pull are two main data stream transmissions in a BDSC system.

In a push system, once an upstream node gets a result, it will immediately push the result data to downstream nodes. Doing like this, the upstream data will be immediately sent to downstream nodes. However, if some downstream nodes are busy or fail, some data will be discarded.

In a pull system, a downstream node requests data from an upstream node. If some data need to be further processed, the upstream node will send the data to the requesting downstream node. Doing like this, the upstream data will be stored in upstream nodes until corresponding downstream nodes request. Some data will wait a long time for further processing and may lose their timeliness.

1.2.3.3 Application Interfaces

An application interface is used to design a data stream graph, a bridge between a user and a BDSC system. Usually, a good application interface is flexible and efficient for users. Currently, most of BDSC systems provide MapReduce-like interfaces, e.g., the Storm system provides Spout and Bolt as an application interface, and a user can design a data stream graph by Spout and Bolt.

Some other BDSC systems provide SQL-like interfaces and graphical user interfaces.

1.2.3.4 High-Availability

State backup and recovery is the main method to achieve high-availability in a BDSC system. There are three main high-availability strategies, i.e., passive standby strategy, active standby strategy, and upstream backup strategy.

In passive standby strategy (see Figure 5), each primary node periodically sends checkpoint data to a backup node. If the primary node fails, the backup node takes over from the last checkpoint. Usually, this strategy will achieve precise recovery.

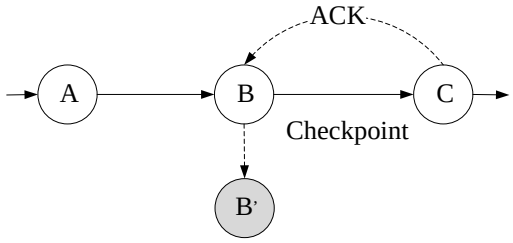

Figure 5. Passive standby

In active standby strategy (see Figure 6), the secondary nodes compute all data stream in parallel with their primaries. Usually, the recovery time of this strategy is the shortest.

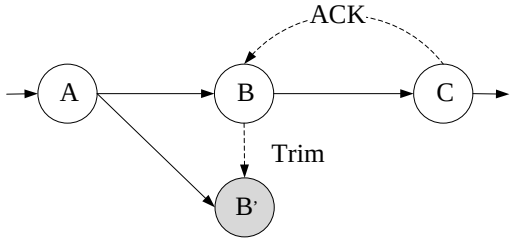

Figure 6. Active standby

In upstream backup strategy (see Figure 7), upstream nodes act as backups for their downstream neighbors by preserving data stream in their output queues while their downstream neighbors compute them. If a node fails, its upstream nodes replay the logged data stream on a recovery node. Usually, the runtime overhead of this strategy is the lowest.

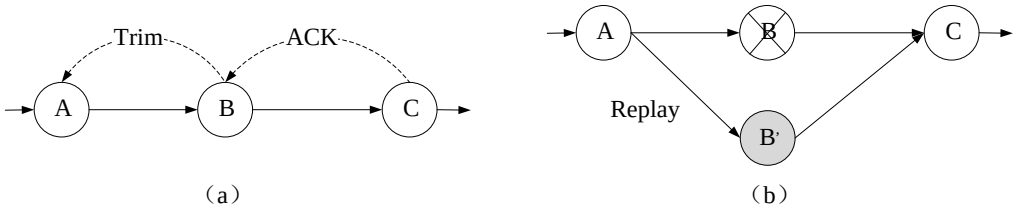

Figure 7. Upstream backup

A comparison of the three main high-availability strategies, i.e., passive standby strategy, active standby strategy, and upstream backup strategy in runtime overhead and recovery time is shown in Figure 8. The recovery time of upstream backup strategy is the longest, while the runtime overhead of passive standby strategy is the greatest.

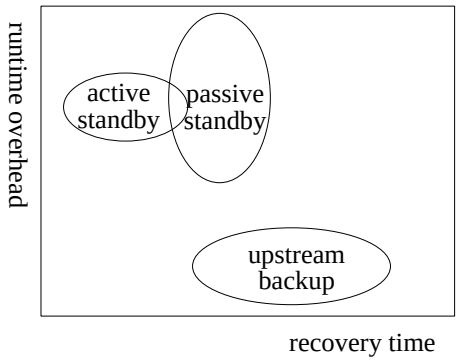

Figure 8. Comparison of high-availability strategies in runtime overhead and recovery time

1.3 Example BDSC Systems

In this section, the system architecture and key technologies of four popular BDSC system instances are presented. These systems are Twitter Storm, Yahoo! S4, and Microsoft TimeStream and Naiad, which are specially designed for big data stream computing.

1.3.1 Twitter Storm

Storm is an open source and distributed big data stream computing system licensed under the Eclipse Public License. Similar to how Hadoop provides a set of general primitives for doing batch

processing, Storm provides a set of general primitives for doing real-time big data computing. Storm platform has the features of simplicity, scalability, fault-tolerance, and so on. It can be used with any programming language, and is easy to set up and operate [1, 20, 21].

1.3.1.1 Task Topology

In big data stream computing environments, the logic for an application is packaged in the form of task topology. Once a task topology is designed and submitted to a system, it will run forever until the user kills it.

A task topology can be described as a directed acyclic graph and comprises of spouts and bolts, as shown in Figure 9. A spout is a source of streams in a task topology, and will read data stream (in tuples) from an external source and emit them into bolts. Spouts can emit more than one data stream. The processing of a data stream in a task topology is done in bolts. Anything can be done by bolts, such as filtering, aggregations, joins, and so on. Some simple functions can be achieved by a bolt, while complex functions will be achieve by many bolts. The logic should be designed by a user. For example, transforming a stream of tweets into a stream of trending images requires at least two steps: a bolt to do a rolling count of retweets for each image, and one or more bolts to stream out the top *n* images. Bolts can also emit more than one stream. Each edge in the directed acyclic graph represents a bolt subscribing to the output stream of some other spout or bolt.

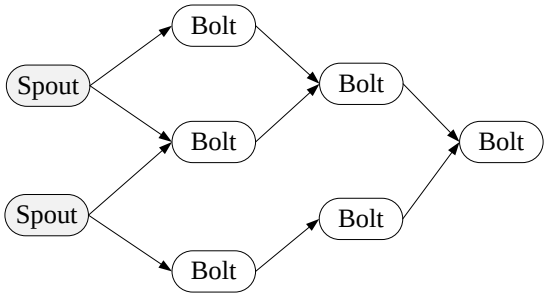

Figure 9. Task topology of Storm

A data stream is an unbounded sequence of tuples that is processed and created in parallel in a

distributed big data stream computing environment. A task topology processes data streams in any complex ways. Repartitioning the streams between each stage of the computation is needed. Task topologies are inherently parallel and run across a cluster of machines. Any vertex in a task topology can be created in many instances. All those vertices will simultaneously process a data stream, and different parts of the topology can be allocated in different machines. A good allocating strategy will greatly improve system performance.

A data stream grouping defines how that stream should be partitioned among the bolt's tasks, spouts and bolts execute in parallel as many tasks across the cluster. There are seven built-in stream groupings in Storm, such as, shuffle grouping, fields grouping, all grouping, global grouping, none grouping, direct grouping, local or shuffle grouping, and a custom stream grouping to meet special needing can also be implemented by the CustomStreamGrouping interface.

1.3.1.2 Fault-Tolerance

Fault-tolerance is an important feature of Storm. If a worker dies, Storm will automatically restart it. If a node dies, the worker will be restarted on another node. In Storm, Nimbus and the Supervisors are designed to be stateless and fail-fast whenever any unexpected situation is encountered, and all state information is stored in Zookeeper server. If Nimbus or the Supervisors die, they will restart like nothing happened. This means you can kill the Nimbus and the Supervisors without affecting the health of the cluster or task topologies.

When a worker dies, the supervisor will restart it. If it continuously fails on startup and is unable to heartbeat to Nimbus, Nimbus will reassign the worker to another machine.

When a machine dies, the tasks assigned to that machine will time-out and Nimbus will reassign those tasks to other machines.

When Nimbus or Supervisor dies, they will restart like nothing happened. No worker processes

are affected by the death of Nimbus or the Supervisors.

1.3.1.3 Reliability

In Storm, the reliability mechanisms guarantee every spout tuple will be fully processed by corresponding topology. It does this by tracking the tree of tuples triggered by every spout tuple and determining when that tree of tuples has been successfully completed. Every topology has a "message timeout" associated with it. If Storm fails to detect that a spout tuple has been completed within that timeout, then it fails the tuple and replays it later.

The reliability mechanisms of Storm are completely distributed, scalable, and fault-tolerant. Storm uses mod hashing to map a spout tuple id to an acker task. Since every tuple carries with it the spout tuple ids of all the trees they exist within, they know which acker tasks to communicate with. When a spout task emits a new tuple, it simply sends a message to the appropriate acker telling it that its task id is responsible for that spout tuple. Then, when an acker sees a tree has been completed, it knows to which task id to send the completion message.

An acker task stores a map from a spout tuple id to a pair of values. The first value is the task id that created the spout tuple which is used later on to send completion messages. The second value is a 64 bit number called the "ack val". The ack val is a representation of the state of the entire tuple tree, no matter how big or how small. It is simply the XOR of all tuple ids that have been created and/or acked in the tree. When an acker task sees that an "ack val" has become 0, then it knows that the tuple tree is completed.

1.3.1.4 Storm Cluster

A Storm cluster is superficially similar to a Hadoop cluster. Whereas on Hadoop you run "MapReduce jobs", on Storm you run "topologies". As shown in Figure 10, there are two kinds of nodes on a Storm cluster, i.e., the master node and the worker nodes.

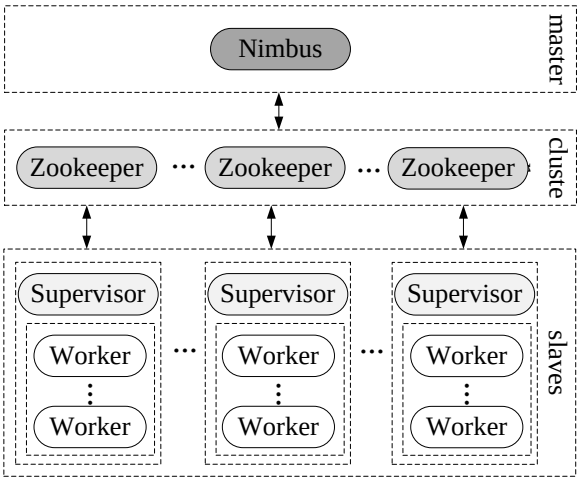

Figure 10. Storm cluster

The master node runs Nimbus node that is similar to Hadoop's “JobTracker”. In Storm, Nimbus node is responsible for distributing code around the cluster, assigning tasks to machines, monitoring for failures, and so on.

Each worker node runs Supervisor node. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology. Usually, a running topology consists of many worker processes spread across many machines.

The coordination between Nimbus and the Supervisors is done through a Zookeeper cluster. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless; all states are kept in Zookeeper server. This means that if you kill the Nimbus or the Supervisors, they will start back up like nothing has happened.

1.3.2 Yahoo! S4

S4 is a general-purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to easily develop applications for computing continuous unbounded streams of big data. The core part of S4 is written in Java. The implementation is modular and pluggable, and S4

applications can be easily and dynamically combined for creating more sophisticated stream processing systems. S4 was initially released by Yahoo! Inc. in October 2010 and is an Apache Incubator project since September 2011. It is licensed under the Apache 2.0 license [22-26].

1.3.2.1 Processing Element

The computing unit of S4 is *processing element* (PE). As shown in Figure 11, each instance of processing element can be identified by four components, i.e., functionality, types of events, keyed attribute, and value of the keyed attribute. Each processing element processes exactly those events which correspond to the value on which it is keyed.

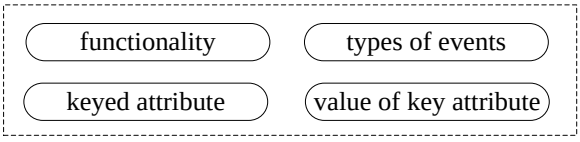

Figure 11. Processing element

A special class of processing elements is the set of keyless processing elements, with no keyed attribute or value. This type of processing elements will process all events of the type with which they are associated. Usually, the keyless processing elements are typically used at the input layer of an S4 cluster, where events are assigned a key.

1.3.2.2 Processing Nodes

Processing nodes (PNs) are the logical hosts to processing elements. Many processing elements are working in processing element container, as shown in Figure 12. A processing node is responsible for event listener, dispatcher events, and emitter output events. In addition, routing model, load balancing model, failover management model, transport protocols, and zookeeper are deployed in communication layer.

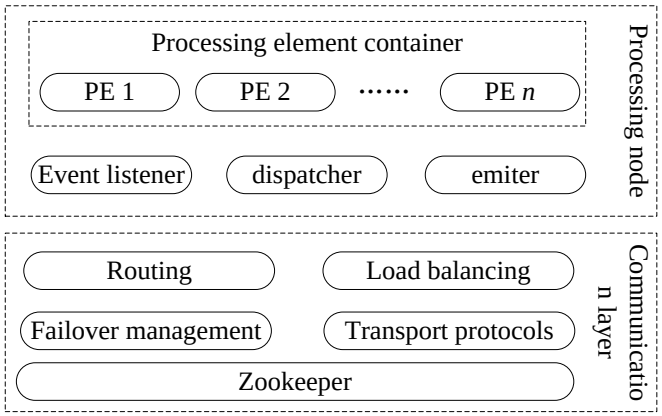

Figure 12. Processing node

All events will be routed to processing nodes by S4 according to a hash function. Every keyed processing element can be mapped to exactly one processing node based on the value of the hash function applied to the value of the keyed attribute of that processing element. However, keyless processing elements may be instantiated on every processing node. The event listener model of processing node will always listen event from S4. If an event is allocated to a processing node, it will be routed to an appropriate processing element within that processing node.

1.3.2.3 Fail-over, Checkpointing, and Recovery Mechanism

In S4, fail-over mechanism will provide a high availability environment for S4. When a node is dead, a corresponding standby node will be used. In order to minimize state loss when a node is dead, a checkpointing and recovery mechanism is employed by S4.

In order to improve the availability of S4 system, it provides a fail-over mechanism to automatically detect failed nodes and redirect data stream to a standby node. If you have *n* partitions and start *m* nodes, with *m*>*n*, you get *m*-*n* standby nodes. For instance, if there are 7 live nodes, 4 partitions available, then 4 of the nodes pick the available partitions in Zookeeper, and the remaining 3 nodes will be standby nodes available. Each active node consistently receives messages for the partition it picked, as shown in Figure 13(a). When Zookeeper detects nodes

failures and notifies the other nodes. As shown in Figure 13(b), the node assigned with partition 1 fails. Unassigned nodes compete for a partition assignment and only 1 of them picks it. Other nodes are notified of the new assignment and can reroute data stream for partition 1, as shown in Figure 13(c).

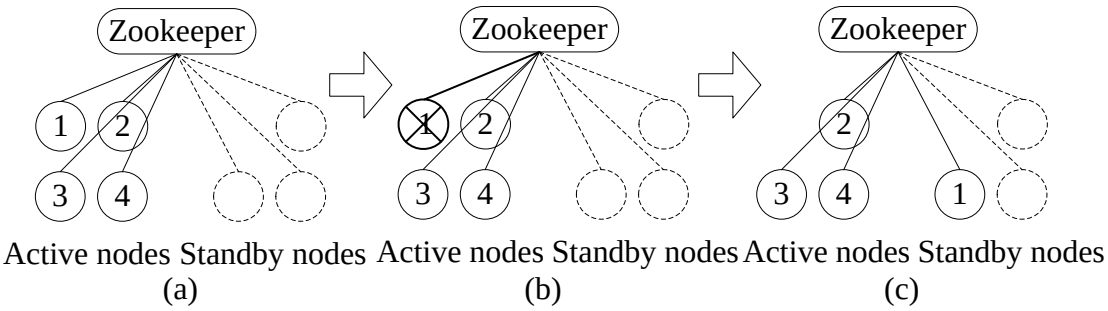

Figure 13. Fail-over mechanism

If a node is unreachable after a session timeout, Zookeeper will identify this node as dead. The session timeout is specified by the client upon connection, and is at minimum twice the heartbeat specified in the Zookeeper ensemble configuration.

In order to minimize state loss when a node is dead, a checkpointing and recovery mechanism is employed by S4. The states of processing elements are periodically checkpointed and stored. Whenever a node fails, the checkpoint information will be used by the recovery mechanism to recover the state of the failed node to the corresponding standby node. Most of the previous state of a failed node can be seen in the corresponding standby node.

1.3.2.4 System Architecture

In S4, a decentralized and symmetric architecture is used; all nodes share the same functionality and responsibilities (see Figure 14). There is no central node with specialized responsibilities. This greatly simplifies deployment and maintenance.

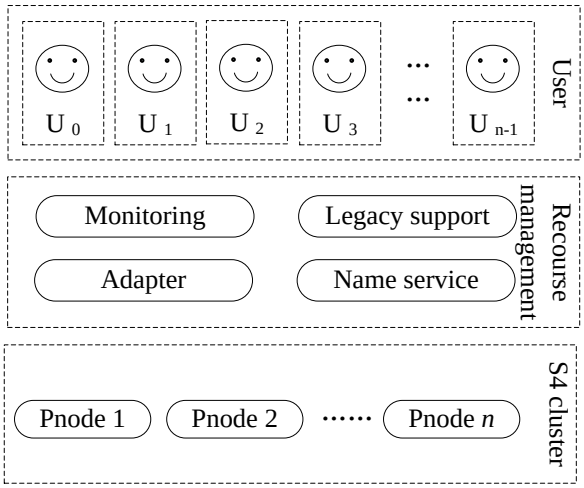

Figure 14. System architecture

A pluggable architecture is used to keep the design as generic and customizable as possible.

1.3.3 Microsoft TimeStream and Naiad

TimeStream and Naiad are two big data stream computing systems of Microsoft.

1.3.3.1 TimeStream

TimeStream is a distributed system designed specifically for low-latency continuous processing of big streaming data on a large cluster of commodity machines, and is based on StreamInsight. TimeStream handles an on-line advertising aggregation pipeline at a rate of 700,000 URLs per second with a 2-second delay [5, 27-30].

(1) Streaming DAG

Streaming DAG is a type of task topology, which can be dynamically reconfigured according to the loading of data stream. All data streams in TimeStream system will be processed in streaming DAG. Each vertex in streaming DAG will be allocated to physical machines for execution. As shown in Figure 15, streaming function f_v of vertex *v* is designed by user. When input data stream *i* is coming, streaming function f_v will process data stream *i*, update *v*'s state from τ to τ', and produce a sequence *o* of output entries as part of the output streams for downstream vertices.

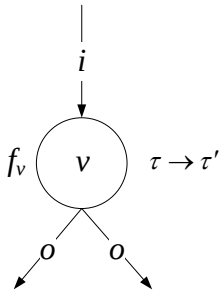

Figure 15. Streaming DAG

A sub-DAG is logically equivalent and can be reduced to one vertex or another sub-DAG. As shown in Figure 16, the sub-DAG comprised of vertices v_2, v_3, v_4, and v_5 (as well as all their edges) is a valid sub-DAG, and can be reduced to a "vertex" with *i* as its input stream and *o* as its output stream.

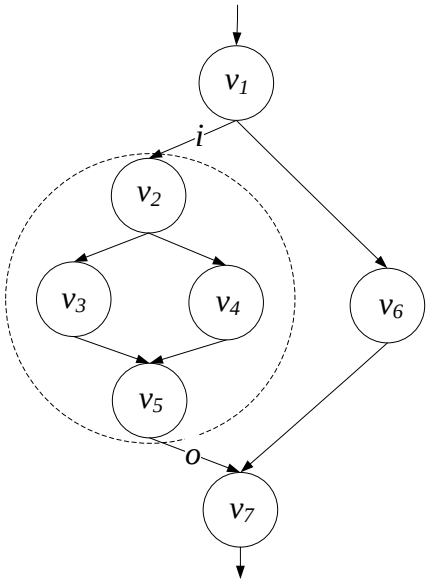

Figure 16. Streaming DAG and sun-DAG

(2) Resilient Substitution

Resilient substitution is an important feature of TimeStream. It is used to dynamic adjust and reconfigure streaming DAG according to the loading change of data stream. There are three types of resilient substitution in TimeStream. (i) A vertex is substituted by another vertex. When a vertex

fails, a new corresponding standby vertex is initiated to replace the failed one and continues execution, possibly on a different machine. (ii) A sub-DAG is substituted by another sub-DAG. When the number of instances of a vertex in sub-DAG needs to be adjusted, a new sub-DAG will replace the old one. For example, as shown in Figure 17, a sub-DAG comprised of vertices v_2, v_3, v_4, and v_5 implements the three stages: hash partitioning, computation, and union. When the load increases, TimeStream can create a new sub-DAG (shown on the left), which uses 4 partitions instead of 2, to replace the original sub-DAG. (iii) A sub-DAG is substituted by a vertex. When the load decreases, there is no need for so many steps to finish a special function, and the corresponding sub-DAG can be substituted by a vertex, as shown in Figure 16.

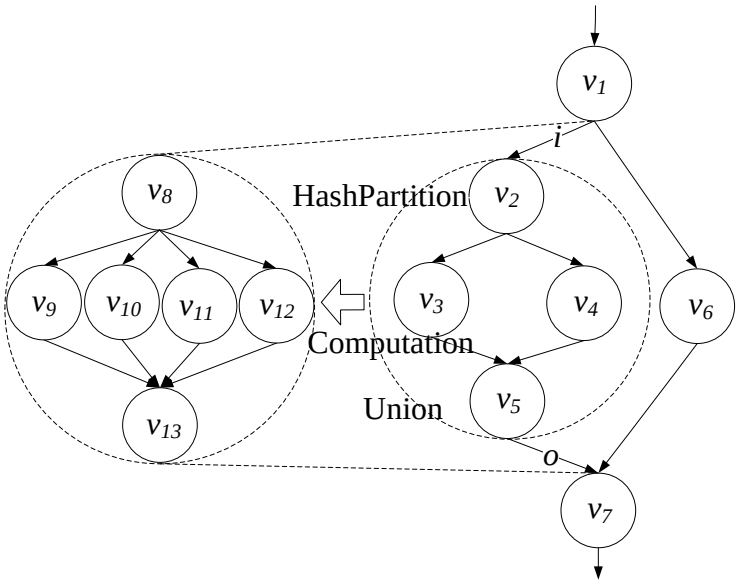

Figure 17. Resilient Substitution

1.3.3.2 Naiad

Naiad is a distributed system for executing data parallel, cyclic dataflow programs. The core part is written in C#. It offers high throughput of batch processors and low latency of stream processors, and is able to perform iterative and incremental computations. Naiad is a prototype implementation of a new computational model timely dataflow [31].

(1) Timely Dataflow

Timely dataflow is a computational model based on directed graphs. The dataflow graph can be a directed acyclic graph, like in other big data stream computing environments. It can also be a directed cyclic graph, the situation of cycles in a dataflow graph is under consideration. In timely dataflow, the timestamps reflect cycle structure in order to distinguish data that arise in different input epochs and loop iterations. The external producer labels each message with an integer epoch, and notifies the input vertex when it will not receive any more messages with a given epoch label. Timely dataflow graphs are directed graphs with the constraint that the vertices are organized into possibly nested loop contexts, with three associated system-provided vertices. Edges entering a loop context must pass through an ingress vertex and edges leaving a loop context must pass through an egress vertex. Additionally, every cycle in the graph must be contained entirely within some loop context, and include at least one feed-back vertex that is not nested within any inner loop con-texts. As show in Figure 18, it shows a single loop context with ingress (‘I’), egress (‘E’), and feedback (‘F’) vertices labeled.

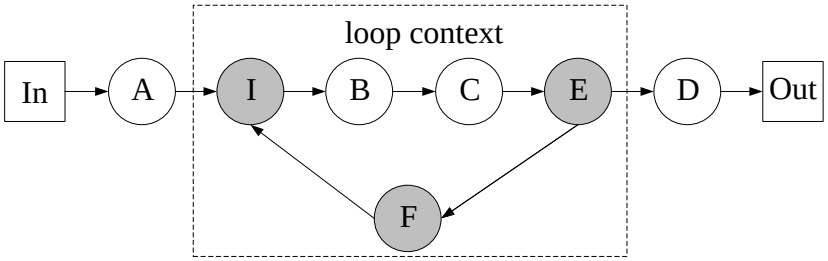

Figure 18. Timely dataflow graph

(2) System Architecture

The system architecture of a Naiad cluster is shown in Figure 19, with a group of processes hosting workers that manage a partition of the timely dataflow vertices. Workers exchange messages locally using shared memory, and remotely using TCP connections between each pair of

processes.

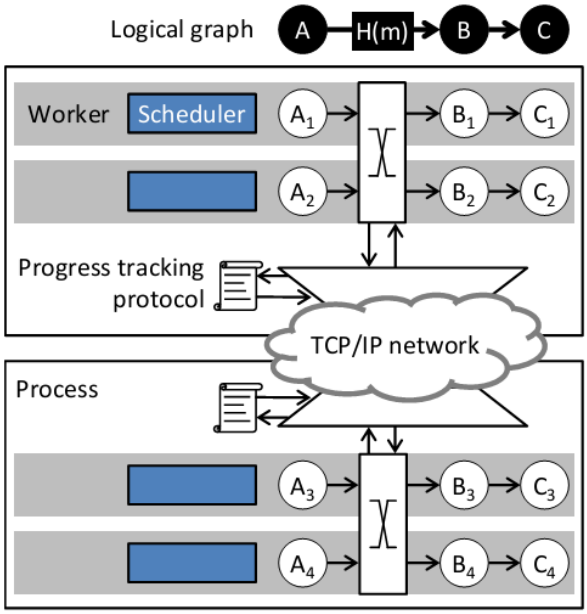

Figure 19. System architecture of a Naiad cluster

A program specifies its timely dataflow graph as a logical graph of stages linked by typed connectors. Each connector optionally has a partitioning function to control the exchange of data between stages. At execution time, Naiad expands the logical graph into a physical graph where each stage is replaced by a set of vertices and each connector by a set of edges. As shown in Figure 19, a logical graph and a corresponding physical graph, where the connector from *A* to *B* has partitioning function $H(m)$ on typed messages m.

Each Naiad worker is responsible for delivering messages and notifications to vertices in its partition of the timely dataflow graph. When faced with multiple runnable actions, workers break ties by delivering messages before notifications, in order to reduce the amount of queued data.

1.4 Future Perspective

In this section, we focus our attention on grand challenges of big data stream computing and the main work we will perform in the near future.

1.4.1 Grand Challenges

Big data stream computing is becoming the fastest and most efficient way to obtain useful knowledge from what is happening now, allowing organizations to react quickly when problems appear or to detect new trends helping to improve their performance. Big data stream computing is needed to manage the data currently generated at an ever increasing rate from such applications as log records or click-streams in web exploring, blogging, twitter posts. In fact, all data generated can be considered as streaming data or as a snapshot of streaming data.

There are some challenges that researchers and practitioners have to deal with in the next few years, such as high scalability, high fault tolerance, high consistency, high load balancing, high throughput, and so on [6, 9, 32-33]. Those challenges arise from the nature of stream data, i.e., data arrive at high speed and must be done under very strict constraints of space and time.

1.4.1.1 High Scalability

High scalability of stream computing can expand to support increasing data stream and meet the QoS of users, or shrink to support decreasing data stream and improve resource utilization. In big data stream computing environments, it is different to achieve high scalability, as the change of data stream is unexpected. The key is that the software changes along with the data stream change, grows along with the increased usage, or shrinks along with the decreased usage. This means scalable programs take up limited space and resources for smaller data needing, but can grow efficiently as more demands are placed on the data stream.

To achieve high scalability in big data stream computing environments, a good scalable system architecture, a good effective resource allocation strategy, and a good data stream computing mode are required.

1.4.1.2 High Fault Tolerance

Highly fault tolerant stream computing can enable a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. Fault-tolerance is particularly sought-after in high-availability or life-critical systems. In big data stream computing environments, it is different to achieve high fault tolerance, as data stream is infinite and real time, and more importantly, most of the data are useless.

To achieve high fault tolerance in big data stream computing environments, a good scalable high fault tolerance strategy is needed. As fault tolerance provides additional resources that allow an application to continue working after a component failure without interruption.

1.4.1.3 High Consistency

Highly consistent stream computing can improve system stability and enhance system efficiency. In big data stream computing environments, it is different to achieve high consistency, as it is hard to decide which nodes should be consistent, and which data are needed.

To achieve high consistency in big data stream computing environments, a good system structure is required. Usually, the master-slave structure is a good choose, as all data are in the master node, and it is easy to achieve highly consistent states.

1.4.1.4 High Load Balancing

Highly load balanced stream computing can make a stream computing system self-adaptive to the changes of data streams, and avoid load shedding. In big data stream computing environments, it is different to achieve high load balancing, as it is impossible to dedicate resources that cover peak loads for 24 hours a day, 7 days a week. Traditionally, stream computing systems use load shedding when the workload exceeds their processing. This employs a trade-off between delivering a low-latency response and ensuring that all incoming data stream are processed.

However, load shedding is not feasible when the variance between peak and average workload is high, and the response should always keep in real-time for user.

To achieve high load balancing in big data stream computing environments, a good distributed computing environment is needed. It should provide a scalable stream computing that automatically streams a partial data stream to a global computing center when local resources become insufficient.

1.4.1.5 High Throughput

High throughput stream computing will improve data stream computing ability by running multiple independent instances of a task topology graph on multiple processors at the same time. In big data stream computing environments, it is different to achieve high throughput, as it is hard to decide how to identify the needing replication sub-graph in task topology graph, to decide the number of replicas, and to decide the fraction of the data stream to assign to each replica.

To achieve high throughput in big data stream computing environments, a good multiple instances replication strategy is needed. Usually, the data stream loading of all instances of all nodes in a task topology graph are equal is a good choose, as the computing ability of all computing nodes are efficient, and it is easy to achieve high throughput states.

1.4.2 On-the-fly Work

Future investigation will focus on the following aspects.

(1) Research on new strategies to optimize task topology graph, such as sub graph partitioning strategy, sub graph replication strategy, sub graph allocating strategy, and to provide a high throughput big data stream computing environment.

(2) Research on a dynamical extensible data stream strategies, such that a data stream can be adjusted according to available resources and the QoS of users, and provide a highly load

balancing big data stream computing environment.

(3) Research on the impact of task topology graph with a cycle, and corresponding task topology graph optimize strategy, resource allocating strategy, and provide a highly adaptive big data stream computing environment.

(4) Research on the architectures for large-scale real-time stream computing environments, such as symmetric architecture and master-slave architecture, and provide a highly consistent big data stream computing environment.

(5) Developing a big data stream computing system with the features of high throughput, high fault tolerance, high consistency, and high scalability, and deploying such a system in a real big data stream computing environment.

1.5 References

[1] Strom, http://storm-project.net/.

[2] Neumeyer L, Robbins B, Nair A, et al, "S4: Distributed Stream Computing Platform," Proc. 10th IEEE International Conference on Data Mining Workshops, ICDMW 2010, IEEE Press, Dec. 2010, pp. 170-177.

[3] Zhao Y and Wu J, "Dache: A data aware caching for big-data applications using the MapReduce framework," Proc. 32nd IEEE Conference on Computer Communications, INFOCOM 2013, IEEE Press, Apr. 2013, pp. 35-39.

[4] Shang W, Jiang Z M, Hemmati H, et al, "Assisting developers of big data analytics applications when deploying on Hadoop clouds," Proc. 35th International Conference on Software Engineering, ICSE 2013, IEEE Press, May 2013, pp. 402-411.

[5] Qian Z P, He Y, Su C Z, et al. TimeStream: Reliable stream computation in the cloud, Proc. 8th ACM European Conference on Computer Systems, EuroSys 2013, Prague, Czech republic, ACM Press, Apr. 2013, 1-14.

[6] Umut A A, Yan C. Streaming big data with self-adjusting computation, Proc. 2013 ACM SIGPLAN

Workshop on Data Driven Functional Programming, Co-located with POPL 2013, DDFP 2013, Rome, Italy, ACM Press, Jan. 2013, 15-18.

[7] Demirkan H and Delen D, "Leveraging the capabilities of service-oriented decision support systems: Putting analytics and big data in cloud," Decision Support Systems, vol. 55(1), Apr. 2013, pp. 412-421.

[8] Albert B, "Mining big data in real time," Informatica (Slovenia), vol. 37(1), Mar. 2013, pp. 15-20.

[9] Lu J, Li D, "Bias correction in a small sample from big data," IEEE Transactions on Knowledge and Data Engineering, vol. 25(11), Nov. 2013, pp. 2658-2663.

[10] Tien J M, "Big data: unleashing information," Journal of Systems Science and Systems Engineering, vol. 22(2), Jun. 2013, pp. 127-151.

[11] Zhang R, Koudas N, Ooi B C, et al, "Streaming multiple aggregations using phantoms," VLDB Journal, vol. 19(4), Apr. 2010, pp. 557-583.

[12] Hirzel M, Andrade H, Gedik B, et al, "IBM streams processing language: analyzing big data in motion," IBM Journal of Research and Development, vol. 57(3/4), May-July 2013, pp. 7:1 - 7:11.

[13] Dayarathna M and Toyotaro S, "Automatic optimization of stream programs via source program operator graph transformations," Distributed and Parallel Databases, vol. 31(4), Dec. 2013, pp. 543-599.

[14] Farhad S M, Ko Y, Burgstaller B, et al, "Orchestration by approximation mapping stream programs onto multicore architectures," Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, ACM Press, Mar. 2011, pp. 357-367.

[15] Schneider S, Hirzel M and Gedik B, "Tutorial: Stream processing optimizations," Proc. 7th ACM International Conference on Distributed Event-Based Systems, DEBS 2013, ACM Press, Jun. 2013, pp. 249-258.

[16] Khandekar R, Hildrum K, Parekh S, et al, "COLA: optimizing stream processing applications via graph partitioning," Proc. 10th ACM/IFIP/USENIX International Conference on Middleware, Middleware 2009, ACM Press, Nov. 2009, pp. 1-20.

[17] Scalosub G, Marbach P, Liebeherr J. Buffer management for aggregated streaming data with packet dependencies, IEEE Transactions on Parallel and Distributed Systems, 2013, 24(3): 439-449.

[18] Malensek M, Pallickara S L, Pallickara S. Exploiting geospatial and chronological characteristics in data

streams to enable efficient storage and retrievals, Future Generation Computer Systems, 2013, 29(4): 1049-1061.

[19] Cugola G, Margara A. Processing flows of information: from data stream to complex event processing, ACM Computing Surveys, 2012, 44(3): 15:1-62.

[20] Storm wiki. [2013.07.16]. http://en.wikipedia.org/wiki/Storm.

[21] Storm Tutorial. [2013.07.16]. https://github.com/nathanmarz/storm/ wiki.

[22] Neumeyer L, Robbins B, Nair A, et al. S4: Distributed stream computing platform, Proc. 10th IEEE International Conference on Data Mining Workshops, ICDMW 2010, Sydney, NSW, Australia, IEEE Press, Dec. 2010, 170-177.

[23] Chauhan J, Chowdhury S A, Makaroff D. Performance evaluation of Yahoo! S4: A first look, Proc. 7th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2012, Victoria, BC, Canada, IEEE Press, Nov. 2012, 58-65.

[24] Simoncelli D, Dusi M, Gringoli F, et al. Scaling out the performance of service monitoring applications with BlockMon, Proc. 14th International Conference on Passive and Active Measurement, PAM 2013, Hong Kong, China, IEEE Press, Mar. 2013, 253-255.

[25] S4, distributed stream computing platform. [2013.07.16]. http:// incubator. apache. org/s4/.

[26] Stream Computing StreamBase Yahoo S4 Borealis Comparis. [2013.07.16]. http://oracle-abc.wikidot. com/zh:stream-computing-streambase-yahoo-s4-borealis-comparison.

[27] Guo Z Y, Sean M D, Yang M, et al. Failure recovery: when the cure is worse than the disease, Proc. 14th USENIX conference on Hot Topics in Operating Systems, USENIX 2013, Santa Ana Pueblo New Mexico, USA, ACM Press, May 2013, 1-6.

[28] Ali M, Chandramouli Badrish, Goldstein J, et al. The extensibility framework in Microsoft StreamInsight, Proc. IEEE 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, IEEE Press, Apr. 2011, 1242-1253.

[29] Chandramouli B, Goldstein J, Barga R, et al. Accurate latency estimation in a distributed event processing system, Proc. IEEE 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, IEEE Press, Apr. 2011, 255-266.

[30] Ali M, Chandramouli B, Fay J, et al. Online visualization of geospatial stream data using the WorldWide telescope, VLDB Endowment, 2011, 4(12): 1379-1382.

[31] Derek G M, Frank M S, Rebecca I, et al. Naiad: a timely dataflow system, Proc. The 24th ACM Symposium on Operating Systems Principles, SOSP 2013, Pennsylvania, USA, ACM Press, Nov. 2013, 439-455.

[32] Garzo A, Benczur A A, Sidlo C I, et al. Real-time streaming mobility analytics, Proc. 2013 IEEE International Conference on Big Data, Big Data 2013, Santa Clara, CA, United states, IEEE Press, Oct. 2013, 697-702.

[33] Zaharia M, Das T, Li H, et al. R Discretized streams: Fault-tolerant streaming computation at scale, Proc. the 24th ACM Symposium on Operating Systems Principles, SOSP 2013, Farmington, PA, United states, ACM Press, Nov. 2013, 423-438.